Memoization

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming and Data Structures with Python

Lecture 21, 07 Nov 2023

Inductive definitions, recursive programs, subproblems

- Factorial
  - $fact(0) = 1$
  - $fact(n) = n \times fact(n-1)$
- Insertion sort
  - $isort([\,]) = [\,]$
  - $isort([x_0, x_1, \ldots, x_n]) = insert(isort([x_0, x_1, \ldots, x_{n-1}]), x_n)$

Inductive definitions, recursive programs, subproblems

- Factorial
  - $fact(0) = 1$
  - $fact(n) = n \times fact(n-1)$
- Insertion sort
  - $isort([\,]) = [\,]$
  - $isort([x_0, x_1, \ldots, x_n]) = insert(isort([x_0, x_1, \ldots, x_{n-1}]), x_n)$

```
def fact(n):
  if n <= 0:
    return(1)
  else:
    return(n * fact(n-1))
```

Inductive definitions, recursive programs, subproblems

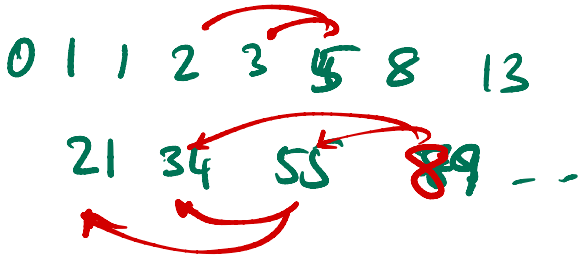
- Factorial
  - $fact(0) = 1$
  - $fact(n) = n \times fact(n-1)$
- Insertion sort
  - $isort([\,]) = [\,]$
  - $isort([x_0, x_1, \ldots, x_n]) = insert(isort([x_0, x_1, \ldots, x_{n-1}]), x_n)$

```
def fact(n):
  if n <= 0:
    return(1)
  else:
    return(n * fact(n-1))
```

- $fact(n-1)$ is a subproblem of $fact(n)$
  - So are $fact(n-2)$, $fact(n-3)$, $\ldots$, $fact(0)$
- $isort([x0, x_1, \ldots, x_{n-1}])$ is a subproblem of $isort([x_0, x_1, \ldots, x_n])$
  - So is $isort([x_0, \ldots, x_j])$ for any $0 < j < n$
- Solution to original problem can be derived by combining solutions to subproblems

Evaluating subproblems

- Fibonacci numbers
  - $fib(0) = 0$
  - $fib(1) = 1$
  - $fib(n) = fib(n-1) + fib(n-2)$



Evaluating subproblems

- Fibonacci numbers
  - $fib(0) = 0$
  - $fib(1) = 1$
  - $fib(n) = fib(n-1) + fib(n-2)$

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) + fib(n-2)
  return(value)
```

Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

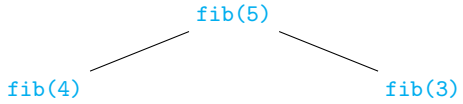
Evaluating `fib(5)`

`fib(5)`

Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

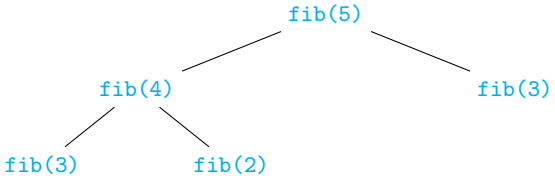
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

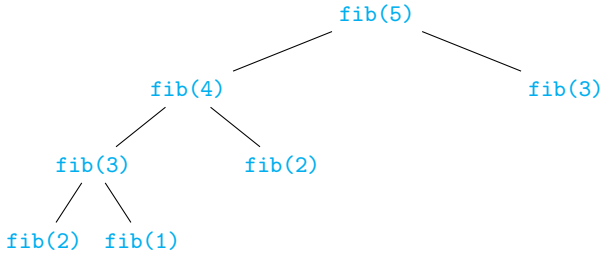
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

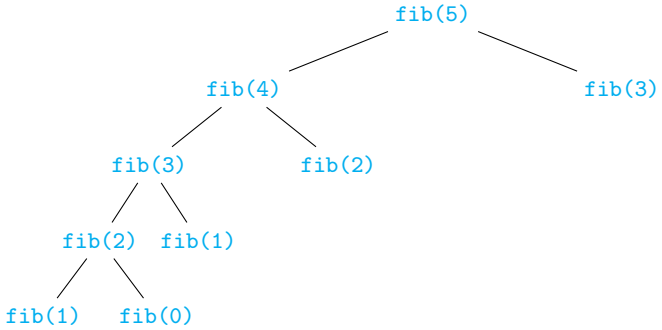
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

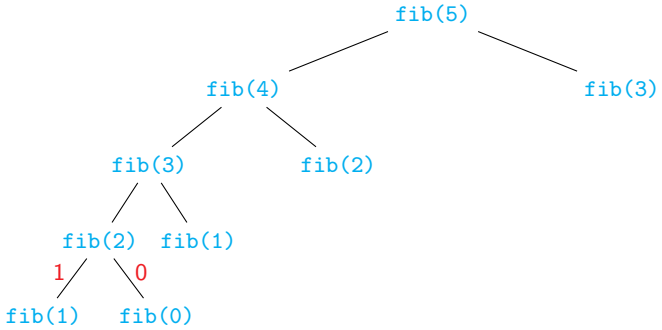
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

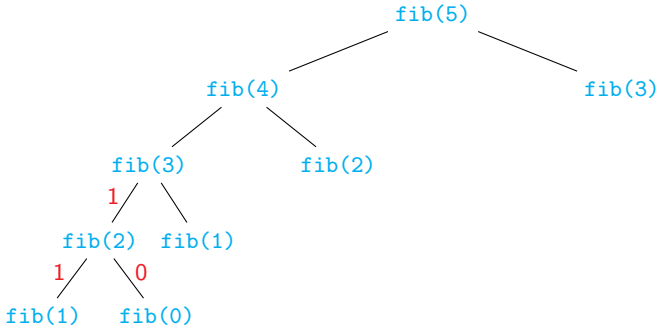
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

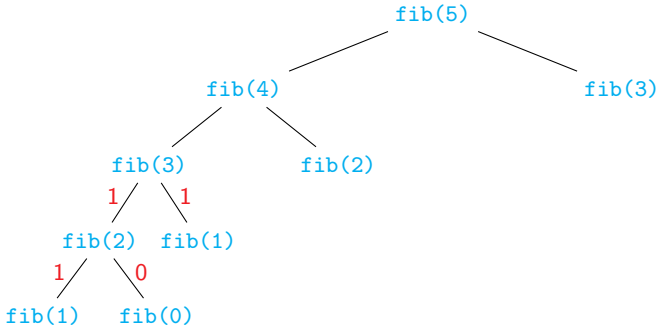
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

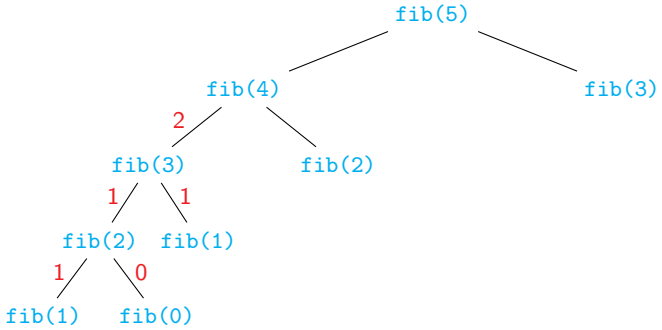
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

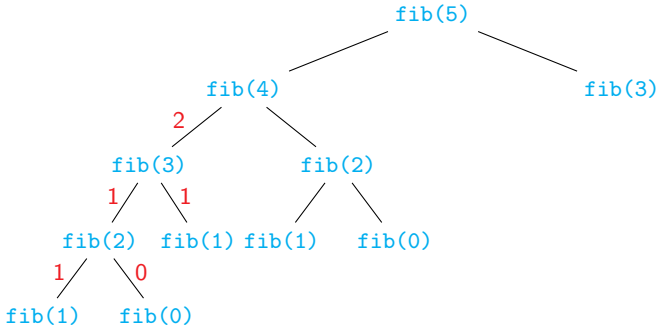
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

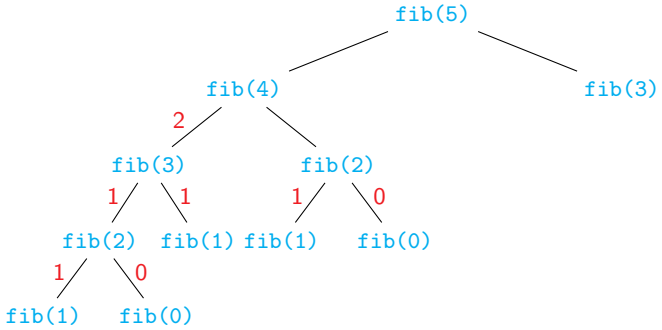
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

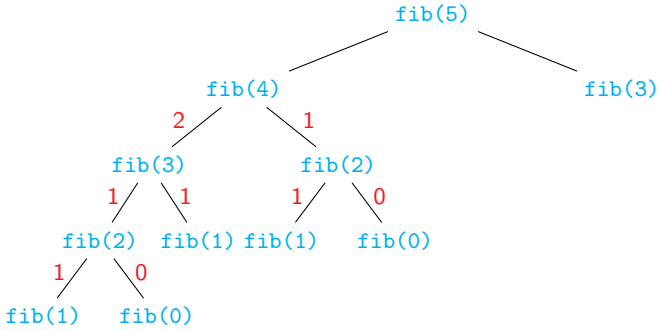
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

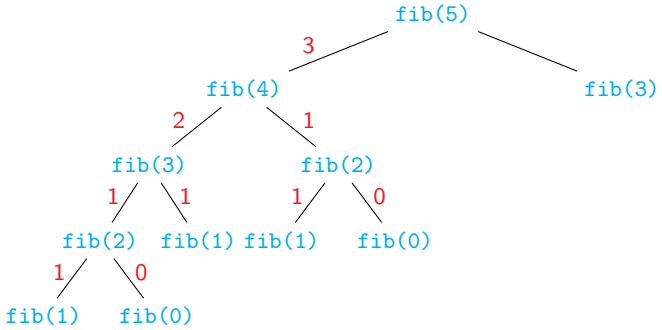
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

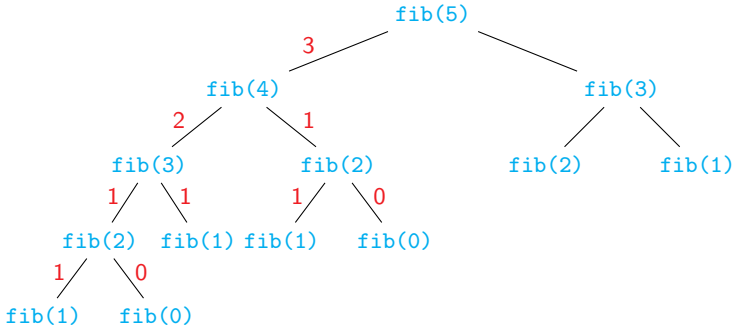
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

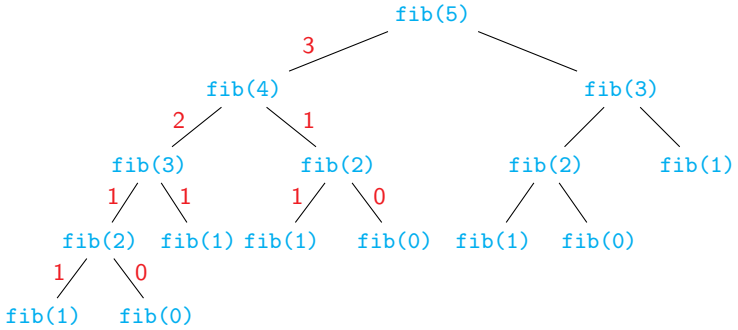
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

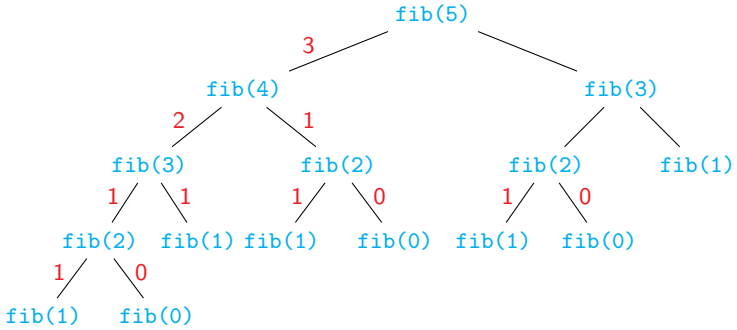
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

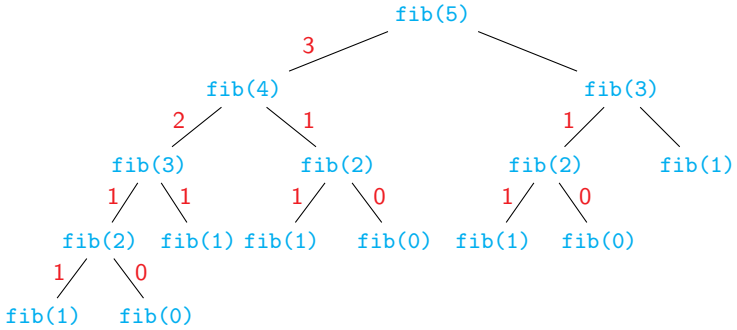
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

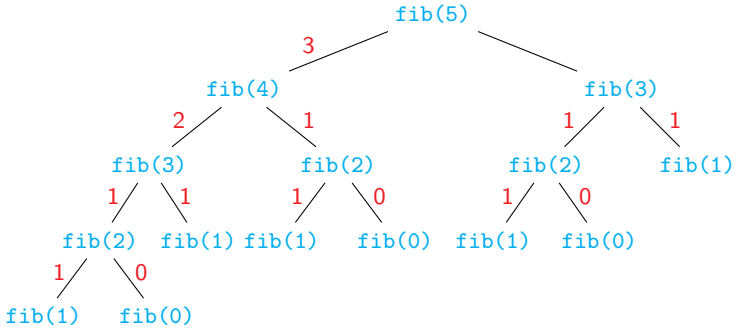
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

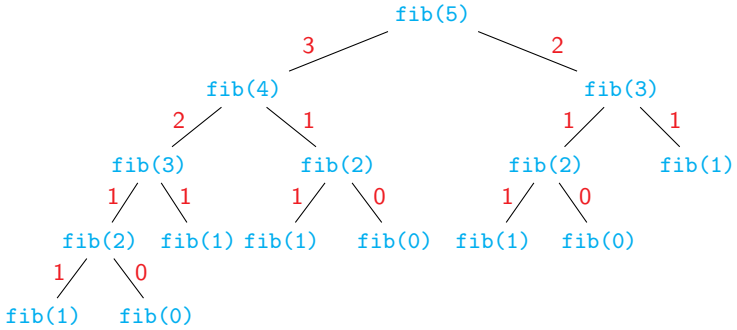
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

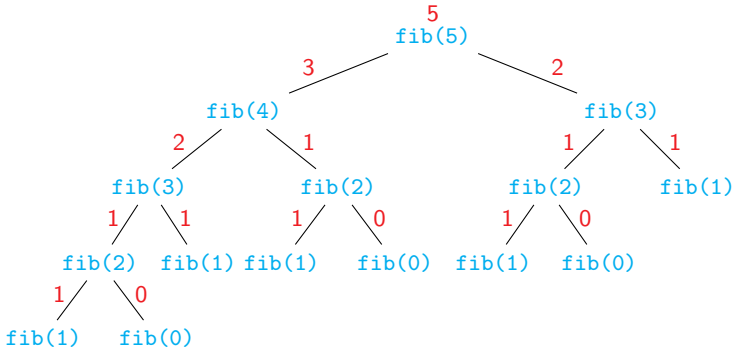
Evaluating `fib(5)`



Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

Evaluating `fib(5)`

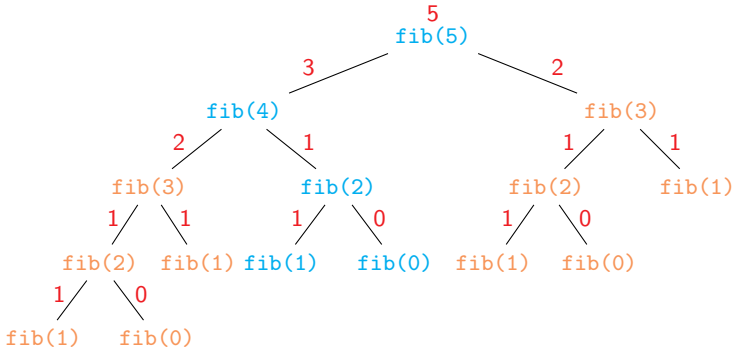


Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

- Wasteful recomputation
- Computation tree grows exponentially

Evaluating `fib(5)`

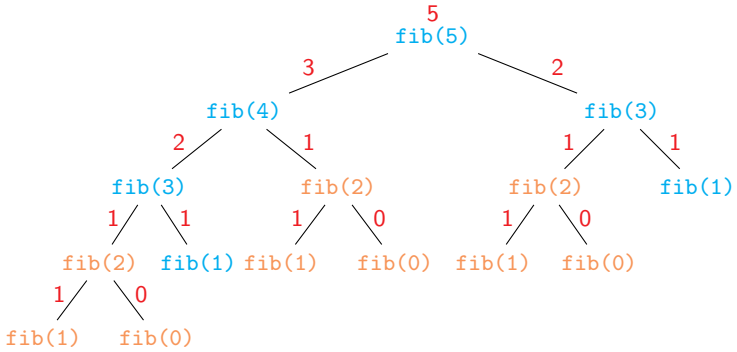


Evaluating subproblems

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

- Wasteful recomputation
- Computation tree grows exponentially

Evaluating `fib(5)`



Evaluating subproblems

- Build a table of values already computed
  - Memory table

Evaluating subproblems

- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before

Evaluating subproblems

- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table

Evaluating subproblems

- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call

Evaluating subproblems

- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear

Evaluating subproblems

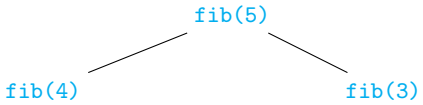
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear

`fib(5)`

k						
fib(k)						

Evaluating subproblems

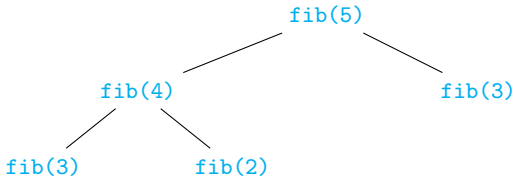
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k						
fib(k)						

Evaluating subproblems

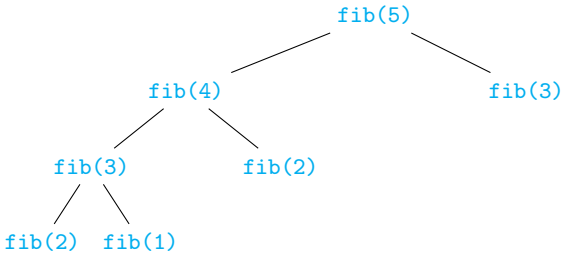
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k						
fib(k)						

Evaluating subproblems

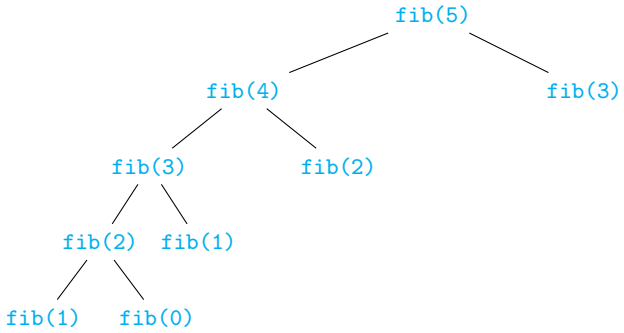
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k						
fib(k)						

Evaluating subproblems

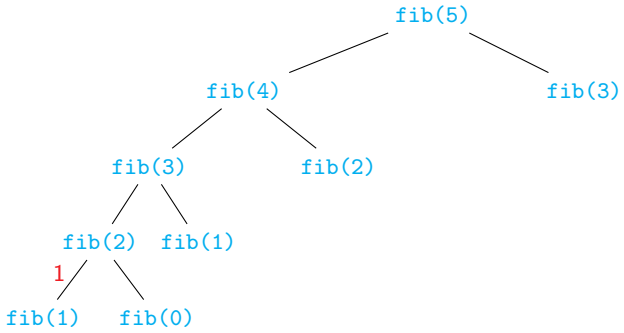
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k						
fib(k)						

Evaluating subproblems

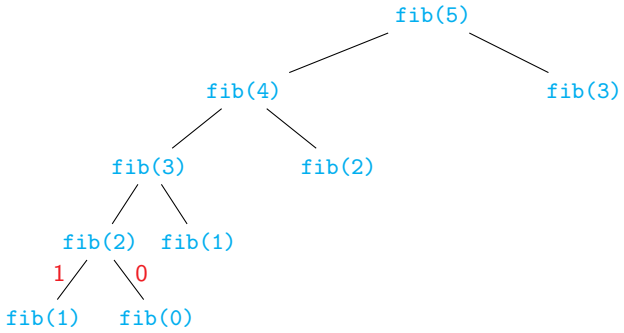
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1					
fib(k)	1					

Evaluating subproblems

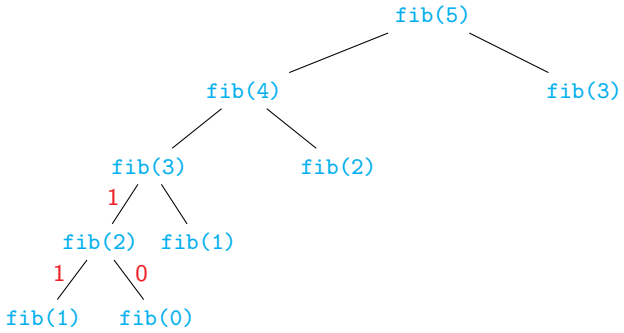
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0				
fib(k)	1	0				

Evaluating subproblems

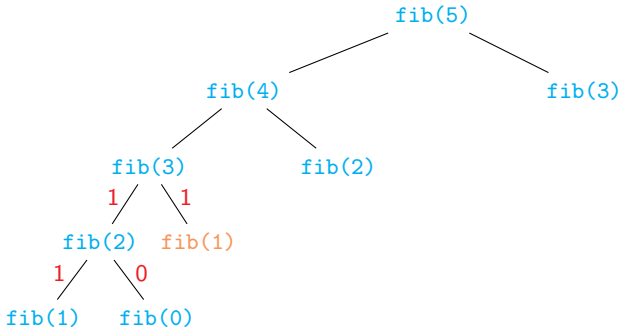
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2			
fib(k)	1	0	1			

Evaluating subproblems

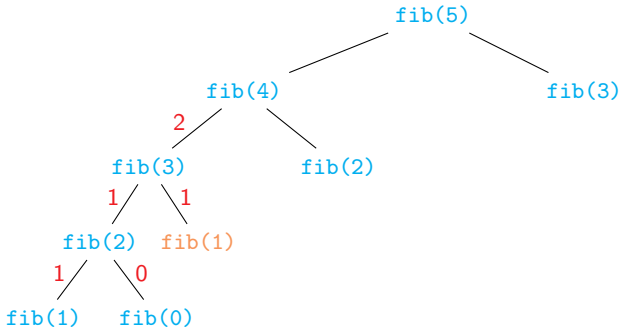
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2			
fib(k)	1	0	1			

Evaluating subproblems

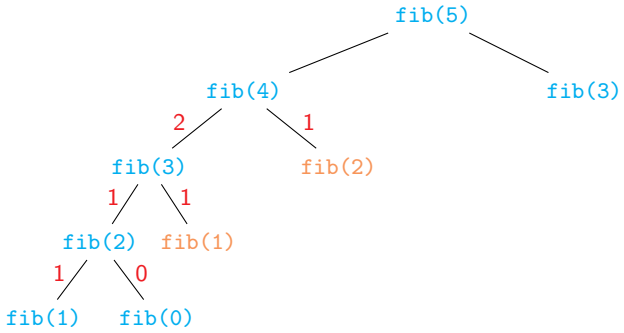
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2	3		
fib(k)	1	0	1	2		

Evaluating subproblems

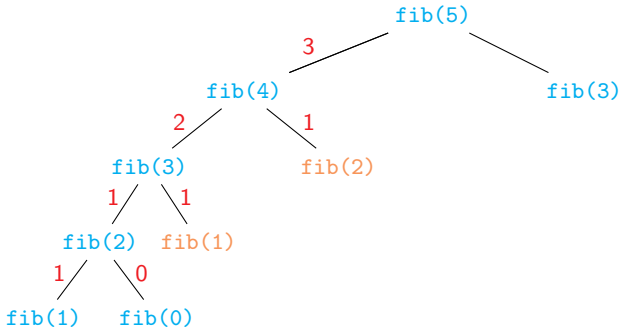
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2	3		
fib(k)	1	0	1	2		

Evaluating subproblems

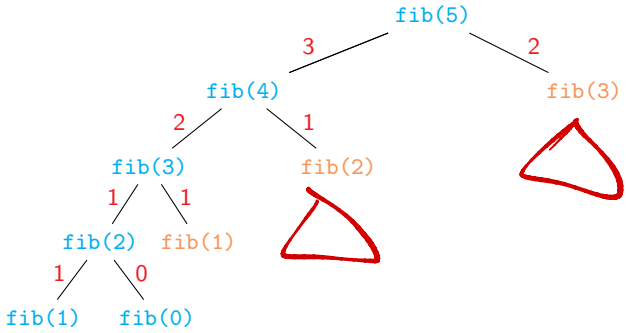
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2	3	4	
fib(k)	1	0	1	2	3	

Evaluating subproblems

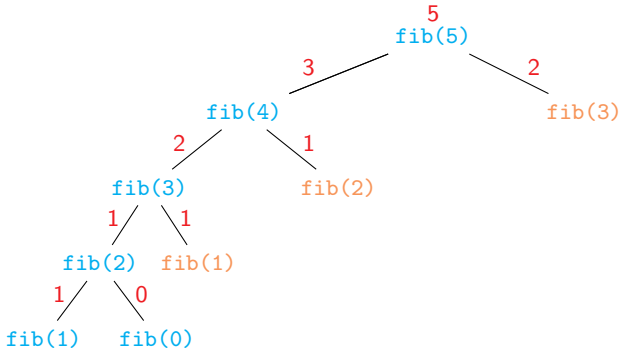
- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2	3	4	
fib(k)	1	0	1	2	3	

Evaluating subproblems

- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



k	1	0	2	3	4	5
fib(k)	1	0	1	2	3	5

Memoizing recursive implmentations

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n-1) + fib(n-2)
  return(value)
```

Memoizing recursive implmentations

fibtable = {}

```
def fib(n):
  if n in fibtable.keys():
    return(fibtable[n])
  if n <= 1:
    value = n
  else:
    value = fib(n-1) + fib(n-2)
  fibtable[n] = value
  return(value)
```

Memoizing recursive implmentations

```
def fib(n):
  if n in fibtable.keys():
    return(fibtable[n])
  if n <= 1:
    value = n
  else:
    value = fib(n-1) + fib(n-2)
  fibtable[n] = value
  return(value)
```

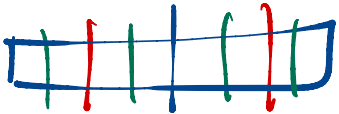
In general

```
def f(x,y,z):
  if (x,y,z) in ftable.keys():
    return(ftable[(x,y,z)])
  recursively compute value
    from subproblems
  ftable[(x,y,z)] = value
  return(value)
```

Divide & Conquer vs functions like Fibonacci

↓

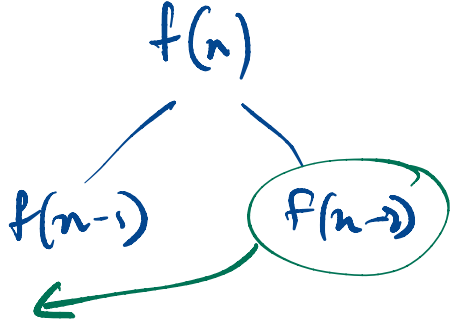
merge sort



Disjoint subproblems

↓

overlapping subproblems



Dynamic Programming

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming and Data Structures with Python

Lecture 21, 07 Nov 2023

Memoizing recursive implementations

```
def fib(n):
  if n in fibtable.keys():
    return(fibtable[n])
  if n <= 1:
    value = n
  else:
    value = fib(n-1) + fib(n-2)
  fibtable[n] = value
  return(value)
```

In general

```
def f(x,y,z):
  if (x,y,z) in ftable.keys():
    return(ftable[(x,y,z)])
  recursively compute value
    from subproblems
  ftable[(x,y,z)] = value
  return(value)
```

Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic

Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic

Evaluating `fib(5)`

`fib(5)`

`fib(4)`

`fib(3)`

`fib(2)`

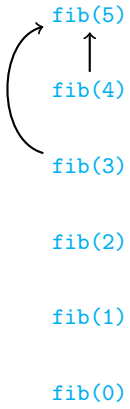
`fib(1)`

`fib(0)`

Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic

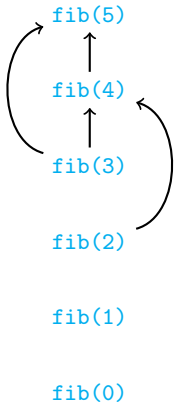
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic

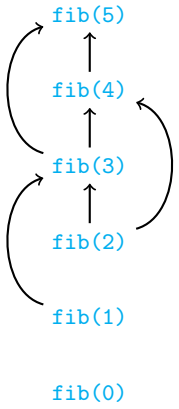
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic

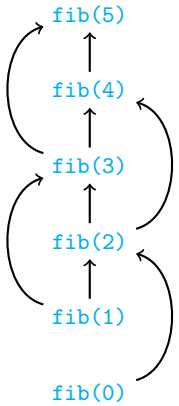
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic

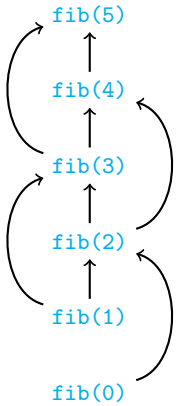
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic
- Solve subproblems in appropriate order

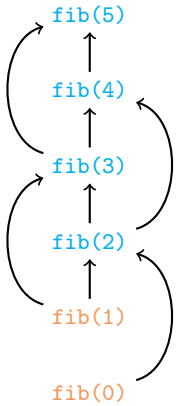
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic
- Solve subproblems in appropriate order
  - Start with base cases — no dependencies

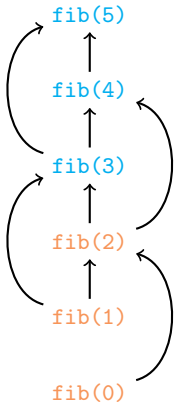
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic
- Solve subproblems in appropriate order
  - Start with base cases — no dependencies
  - Evaluate a value after all its dependencies are available

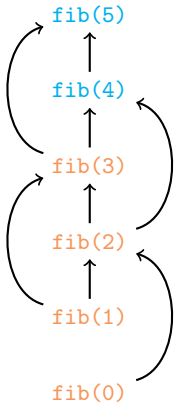
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic
- Solve subproblems in appropriate order
  - Start with base cases — no dependencies
  - Evaluate a value after all its dependencies are available

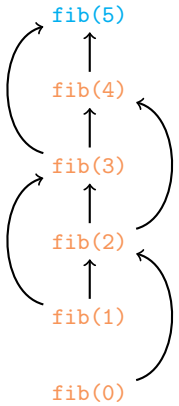
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic
- Solve subproblems in appropriate order
  - Start with base cases — no dependencies
  - Evaluate a value after all its dependencies are available

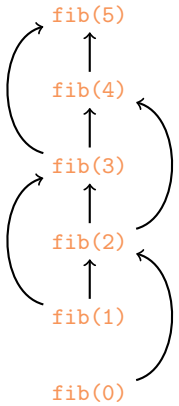
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic
- Solve subproblems in appropriate order
  - Start with base cases — no dependencies
  - Evaluate a value after all its dependencies are available

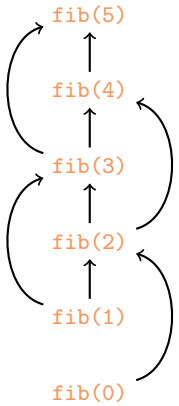
Evaluating `fib(5)`



Dynamic programming

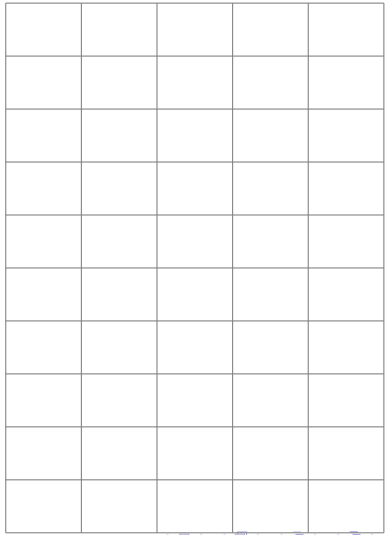
- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic
- Solve subproblems in appropriate order
  - Start with base cases — no dependencies
  - Evaluate a value after all its dependencies are available
  - Fill table iteratively
  - Never need to make a recursive call

Evaluating `fib(5)`



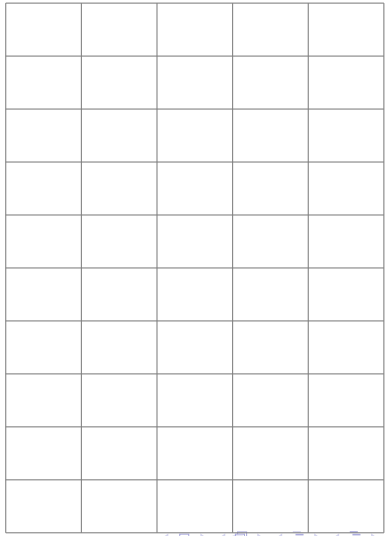
Grid paths

- Rectangular grid of one-way roads



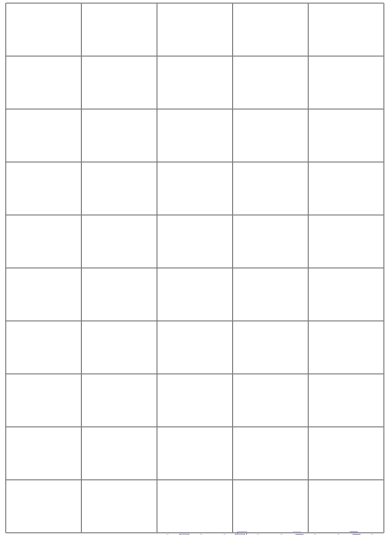
Grid paths

- Rectangular grid of one-way roads
- Can only go up and right



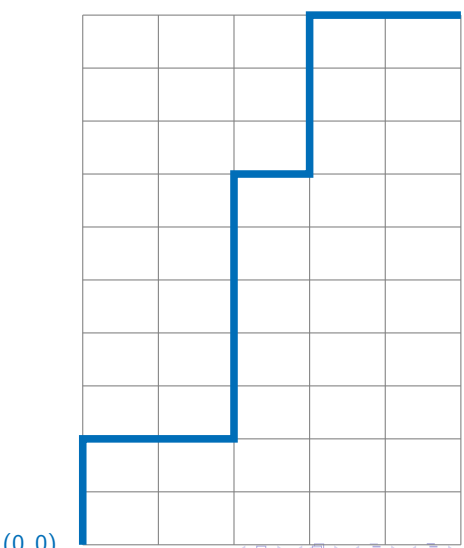
Grid paths

- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from $(0, 0)$ to (m, n)?



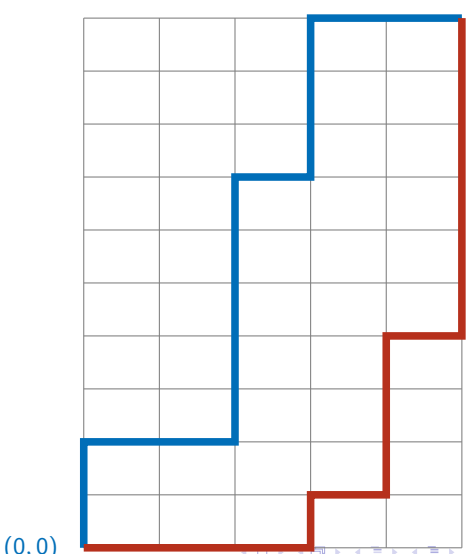
Grid paths

- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from $(0, 0)$ to (m, n)?



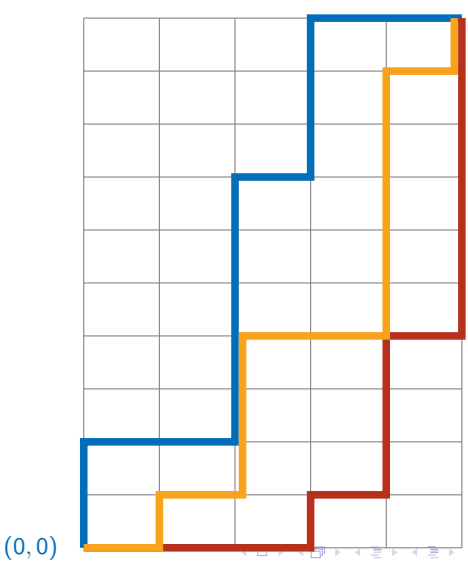
Grid paths

- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from $(0, 0)$ to (m, n)?



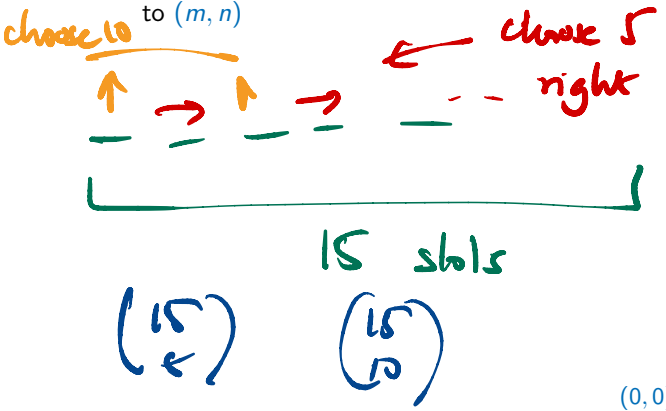
Grid paths

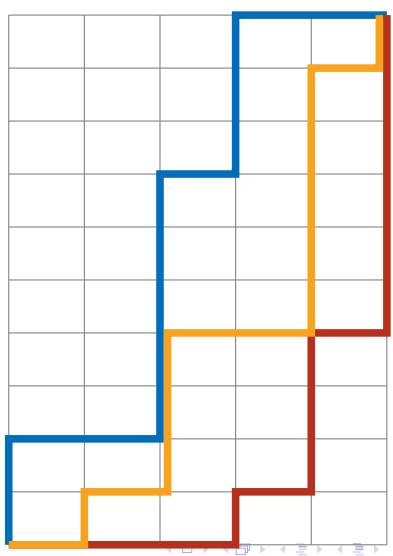
- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from $(0, 0)$ to (m, n)?



Combinatorial solution

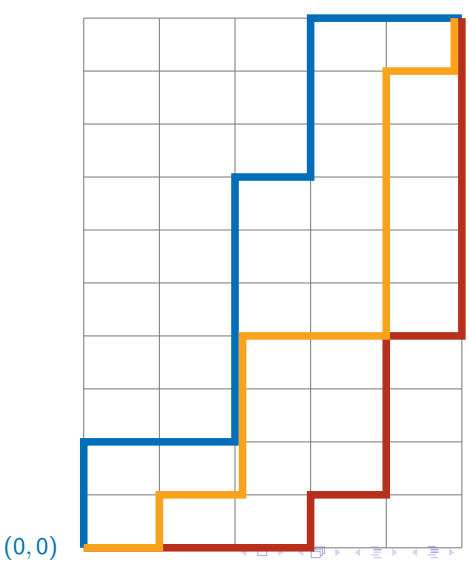
- Every path from $(0, 0)$ to $(5, 10)$ has 15 segments
  - In general $m+n$ segments from $(0, 0)$ to (m, n)





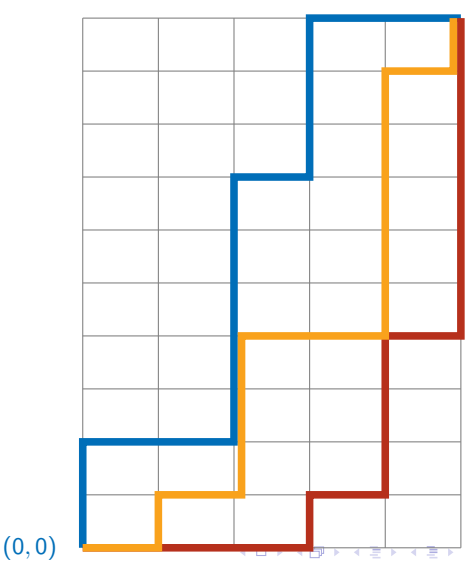
Combinatorial solution

- Every path from $(0, 0)$ to $(5, 10)$ has 15 segments
  - In general $m+n$ segments from $(0, 0)$ to (m, n)
- Out of 15, exactly 5 are right moves, 10 are up moves



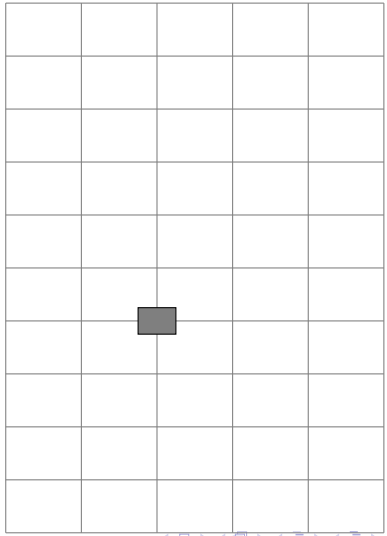
Combinatorial solution

- Every path from $(0, 0)$ to $(5, 10)$ has 15 segments
  - In general $m+n$ segments from $(0, 0)$ to (m, n)
- Out of 15, exactly 5 are right moves, 10 are up moves
- Fix the positions of the 5 right moves among the 15 positions overall
  - $\binom{15}{5} = \frac{15!}{10! \cdot 5!} = 3003$
  - Same as $\binom{15}{10}$ — fix the 10 up moves



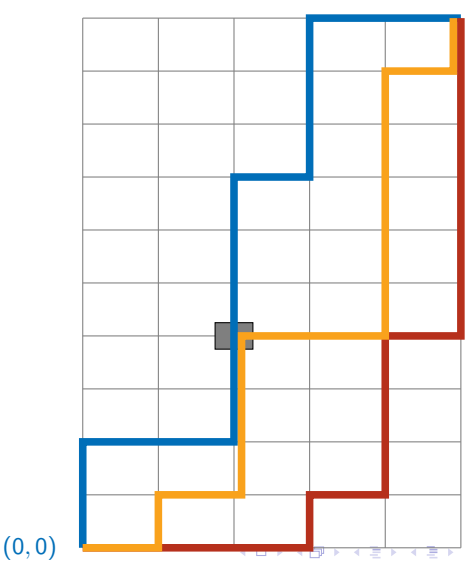
Holes

- What if an intersection is blocked?
  - For instance, $(2, 4)$



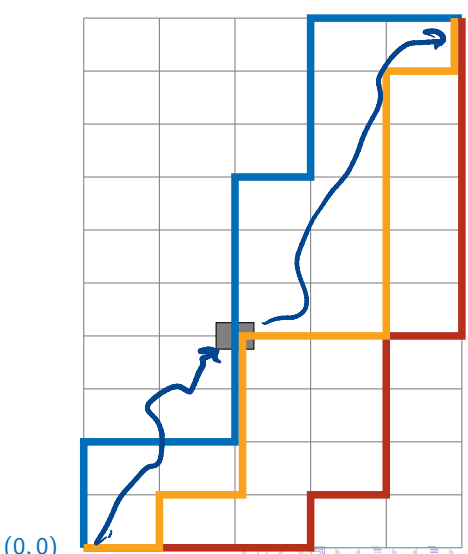
Holes

- What if an intersection is blocked?
  - For instance, $(2, 4)$
- Need to discard paths passing through $(2, 4)$
  - Two of our earlier examples are invalid paths



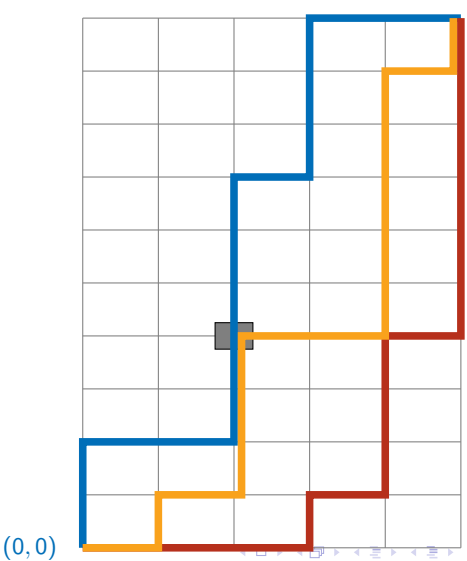
Combinatorial solution for holes

- Discard paths passing through $(2, 4)$



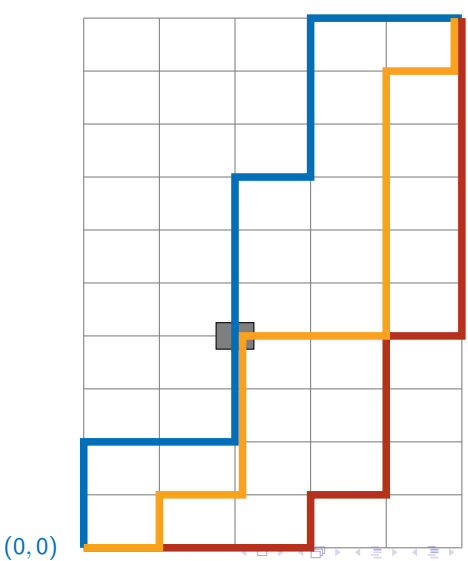
Combinatorial solution for holes

- Discard paths passing through $(2, 4)$
- Every path via $(2, 4)$ combines a path from $(0, 0)$ to $(2, 4)$ with a path from $(2, 4)$ to $(5, 10)$
  - Count these separately
  - $\binom{2+4}{2} = 15$ paths $(0, 0)$ to $(2, 4)$
  - $\binom{3+6}{3} = 84$ paths $(2, 4)$ to $(5, 10)$



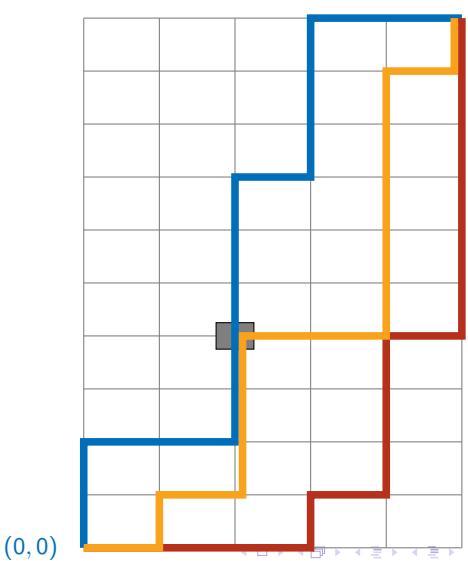
Combinatorial solution for holes

- Discard paths passing through $(2, 4)$
- Every path via $(2, 4)$ combines a path from $(0, 0)$ to $(2, 4)$ with a path from $(2, 4)$ to $(5, 10)$
  - Count these separately
  - $\binom{2+4}{2} = 15$ paths $(0, 0)$ to $(2, 4)$
  - $\binom{3+6}{3} = 84$ paths $(2, 4)$ to $(5, 10)$
- $15 \times 84 = 1260$ paths via $(2, 4)$



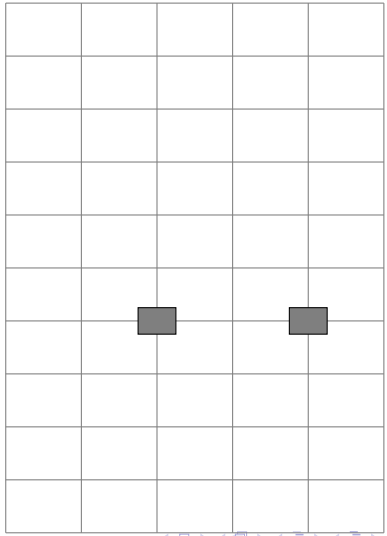
Combinatorial solution for holes

- Discard paths passing through $(2, 4)$
- Every path via $(2, 4)$ combines a path from $(0, 0)$ to $(2, 4)$ with a path from $(2, 4)$ to $(5, 10)$
  - Count these separately
  - $\binom{2+4}{2} = 15$ paths $(0, 0)$ to $(2, 4)$
  - $\binom{3+6}{3} = 84$ paths $(2, 4)$ to $(5, 10)$
- $15 \times 84 = 1260$ paths via $(2, 4)$
- $3003 - 1260 = 1743$ valid paths avoiding $(2, 4)$



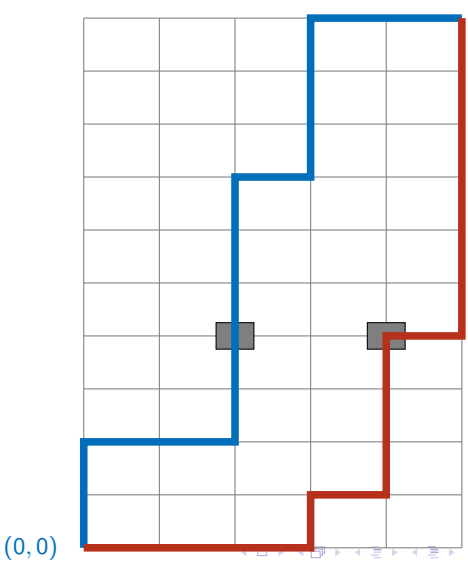
More holes

- What if two intersections are blocked?



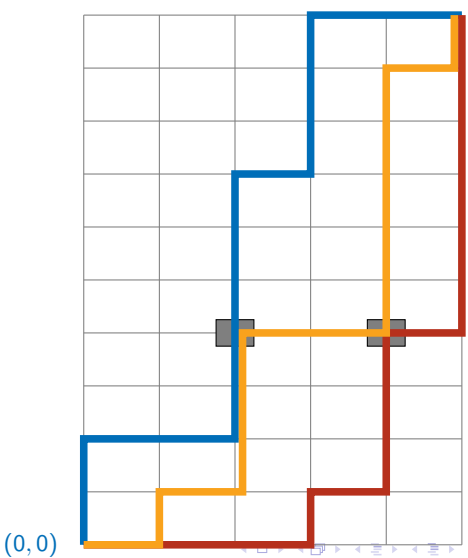
More holes

- What if two intersections are blocked?
- Discard paths via $(2, 4)$, $(4, 4)$



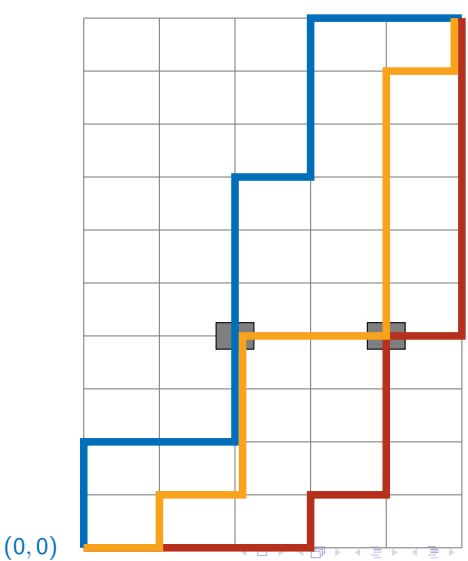
More holes

- What if two intersections are blocked?
- Discard paths via $(2, 4)$, $(4, 4)$
  - Some paths are counted twice



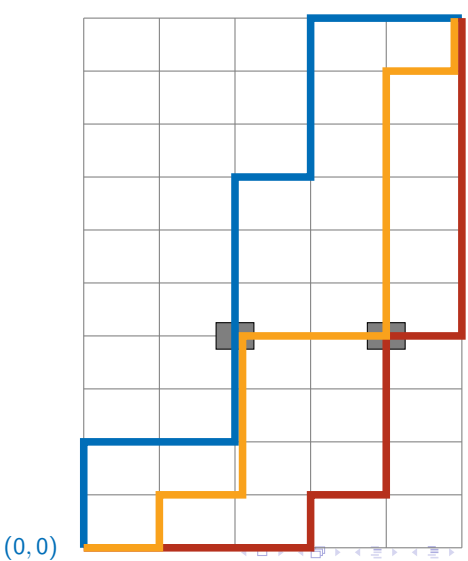
More holes

- What if two intersections are blocked?
- Discard paths via $(2, 4)$, $(4, 4)$
  - Some paths are counted twice
- Add back the paths that pass through both holes



More holes

- What if two intersections are blocked?
- Discard paths via $(2, 4)$, $(4, 4)$
  - Some paths are counted twice
- Add back the paths that pass through both holes
- Inclusion-exclusion — counting is messy



Inductive formulation

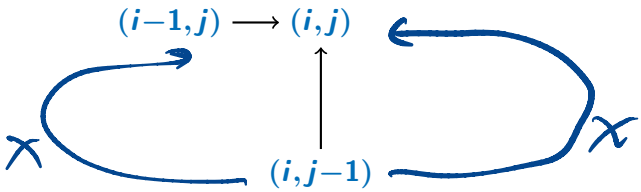
- How can a path reach (i, j)

Inductive formulation

- How can a path reach (i, j)
  - Move up from $(i, j-1)$

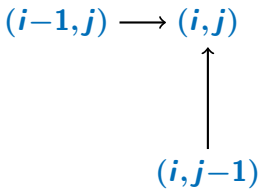
Inductive formulation

- How can a path reach (i, j)
  - Move up from $(i, j-1)$
  - Move right from $(i-1, j)$



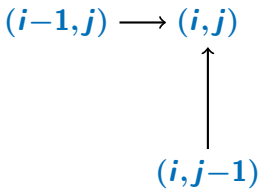
Inductive formulation

- How can a path reach (i, j)
  - Move up from $(i, j-1)$
  - Move right from $(i-1, j)$
- Each path to these neighbours extends to a unique path to (i, j)



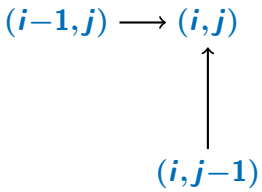
Inductive formulation

- How can a path reach (i, j)
  - Move up from $(i, j-1)$
  - Move right from $(i-1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
  - $P(i, j) = P(i-1, j) + P(i, j-1)$



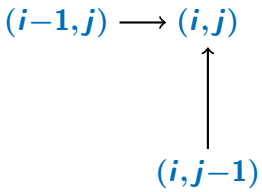
Inductive formulation

- How can a path reach (i, j)
  - Move up from $(i, j-1)$
  - Move right from $(i-1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
  - $P(i, j) = P(i-1, j) + P(i, j-1)$
  - $P(0, 0) = 1$ — base case



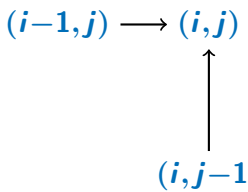
Inductive formulation

- How can a path reach (i, j)
  - Move up from $(i, j-1)$
  - Move right from $(i-1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
  - $P(i, j) = P(i-1, j) + P(i, j-1)$
  - $P(0, 0) = 1$ — base case
  - $P(i, 0) = P(i-1, 0)$ — bottom row



Inductive formulation

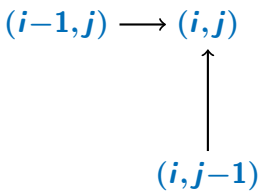
- How can a path reach (i,j)
  - Move up from $(i,j-1)$
  - Move right from $(i-1,j)$
- Each path to these neighbours extends to a unique path to (i,j)
- Recurrence for $P(i,j)$, number of paths from $(0,0)$ to (i,j)
  - $P(i,j) = P(i-1,j) + P(i,j-1)$
  - $P(0,0) = 1$ — base case
  - $P(i,0) = P(i-1,0)$ — bottom row
  - $P(0,j) = P(0,j-1)$ — left column



$P(i,j) = P(\max(i-1,0), j) + P(i, \max(j-1,0))$

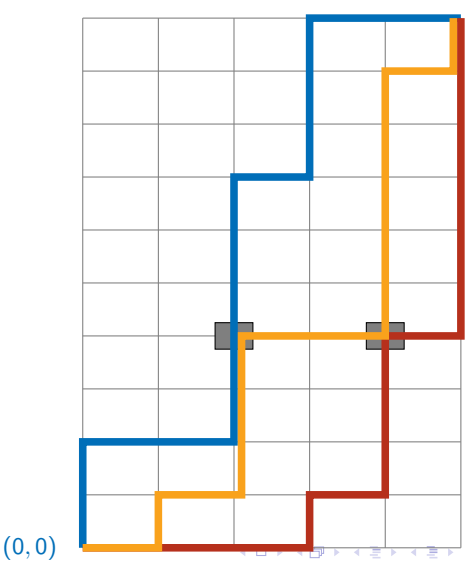
Inductive formulation

- How can a path reach (i, j)
  - Move up from $(i, j-1)$
  - Move right from $(i-1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
  - $P(i, j) = P(i-1, j) + P(i, j-1)$
  - $P(0, 0) = 1$ — base case
  - $P(i, 0) = P(i-1, 0)$ — bottom row
  - $P(0, j) = P(0, j-1)$ — left column
- $P(i, j) = 0$ if there is a hole at (i, j)



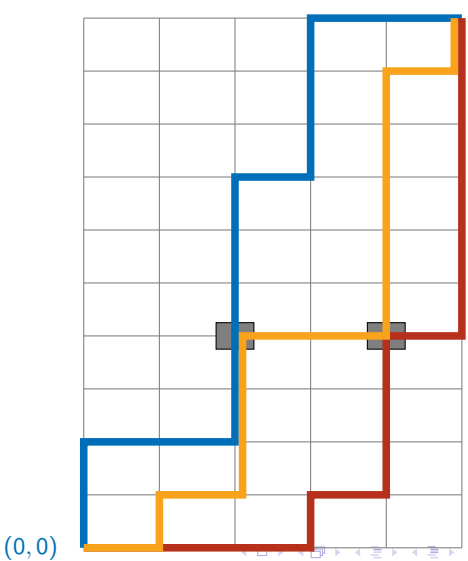
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly



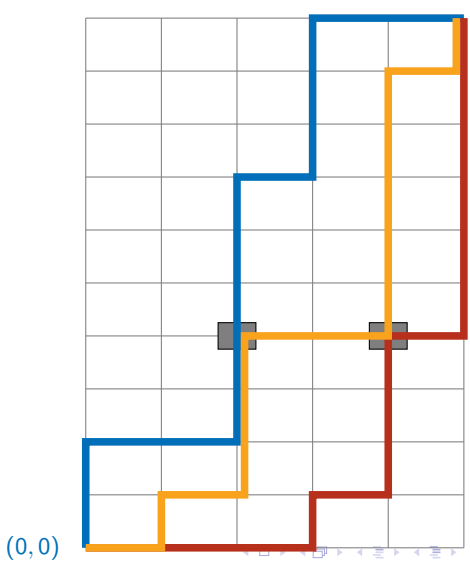
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly
  - $P(5,10)$ requires $P(4,10)$, $P(5,9)$



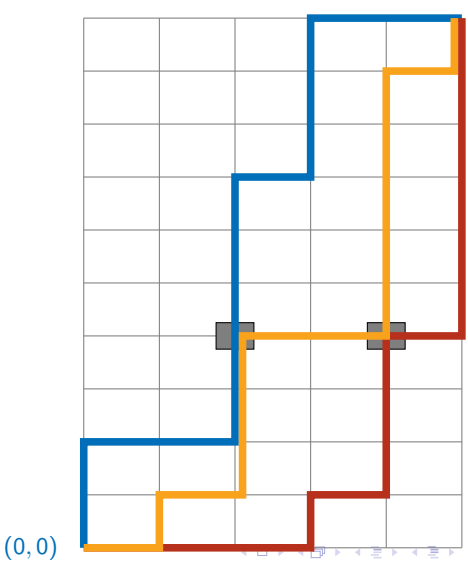
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly
  - $P(5,10)$ requires $P(4,10)$, $P(5,9)$
  - Both $P(4,10)$, $P(5,9)$ require $P(4,9)$



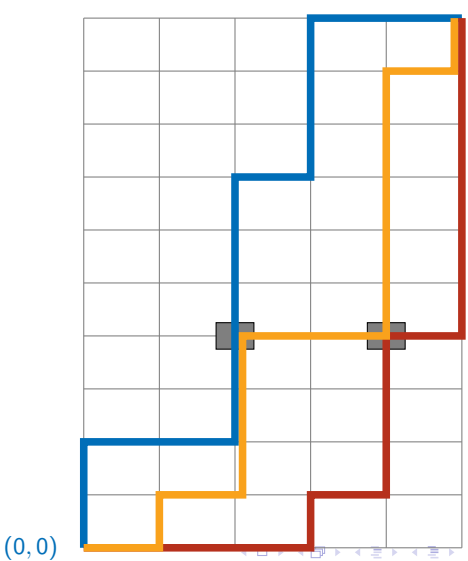
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly
  - $P(5,10)$ requires $P(4,10)$, $P(5,9)$
  - Both $P(4,10)$, $P(5,9)$ require $P(4,9)$
- Use memoization ...



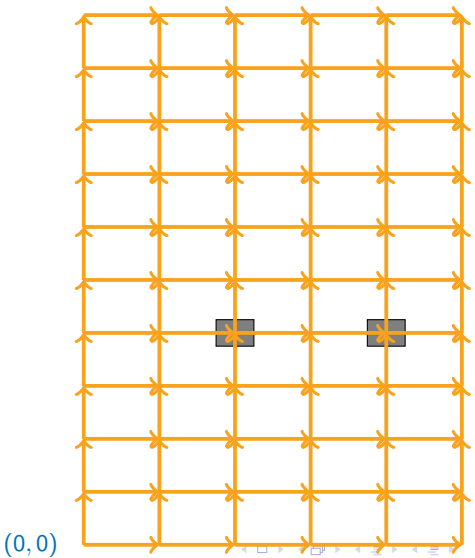
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly
  - $P(5,10)$ requires $P(4,10)$, $P(5,9)$
  - Both $P(4,10)$, $P(5,9)$ require $P(4,9)$
- Use memoization ...
- ... or find a suitable order to compute the subproblems



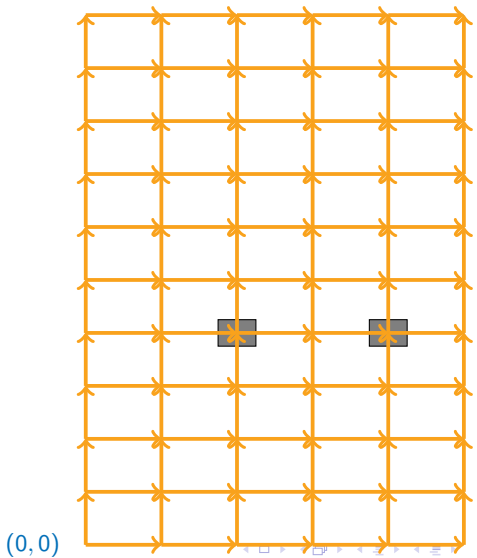
Dynamic programming

- Identify subproblem structure



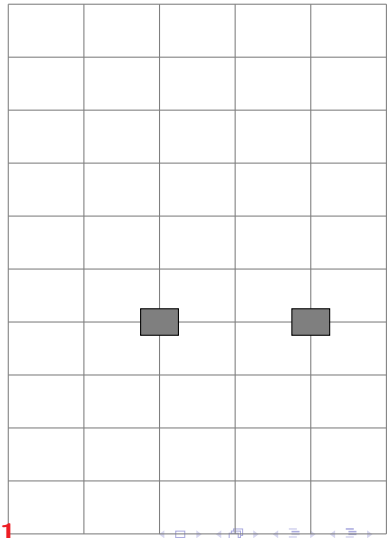
Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies



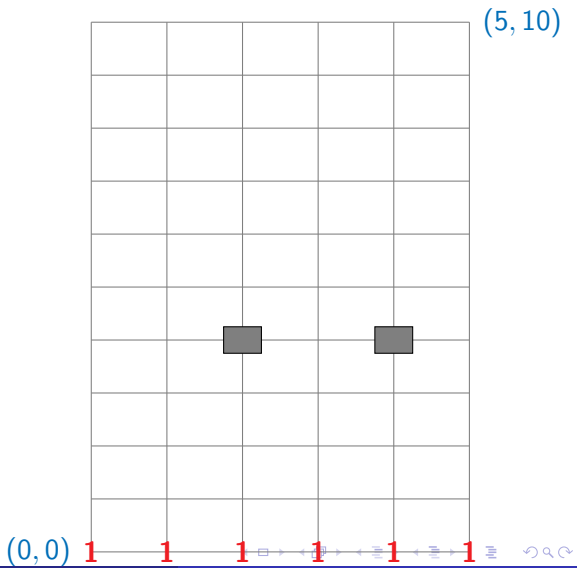
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$



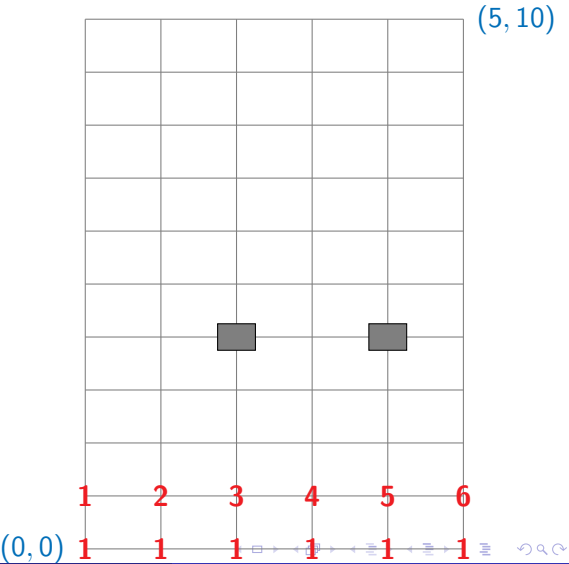
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row



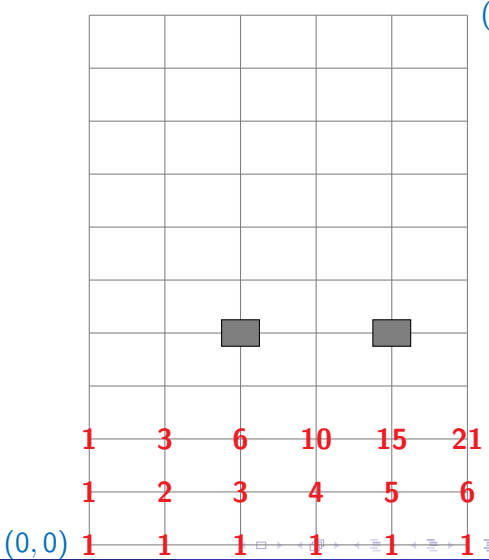
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row



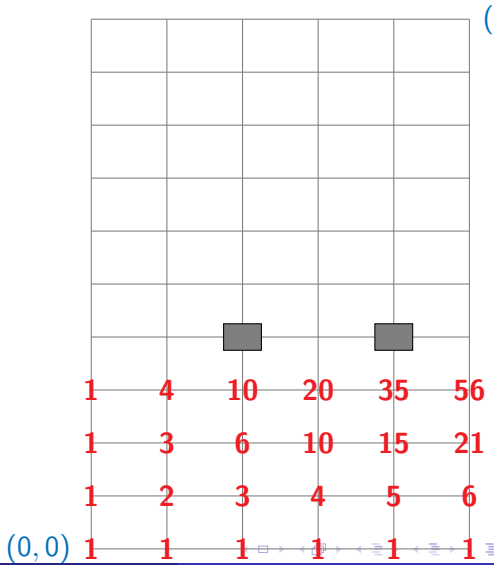
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row



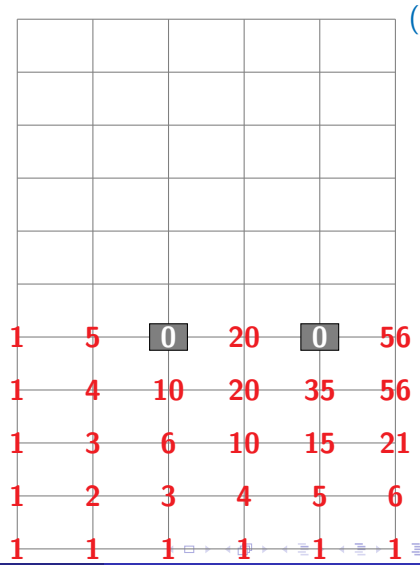
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row



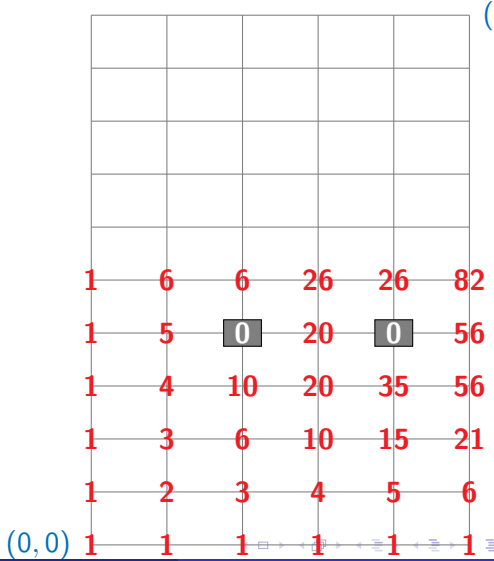
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row



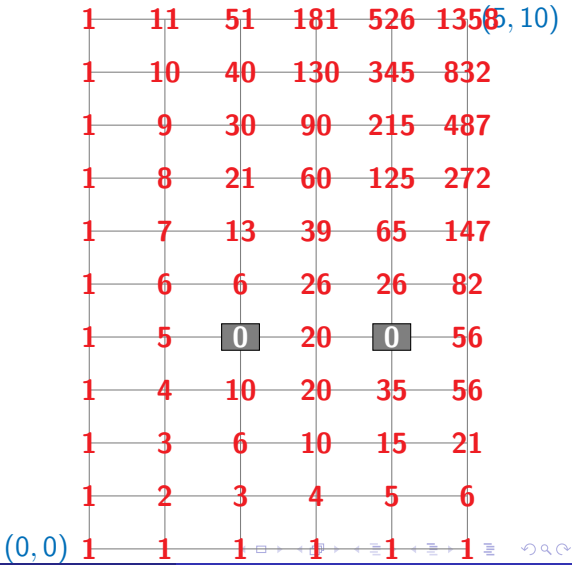
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row



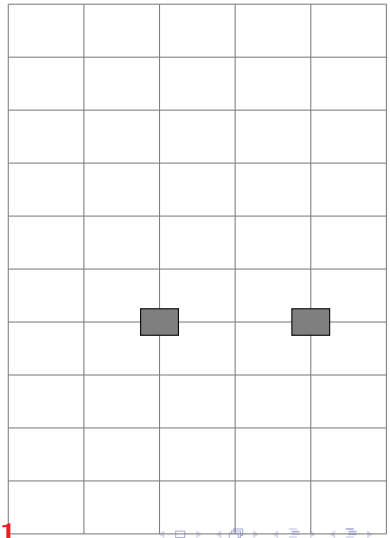
Dynamic programming

- Identify subproblem structure
- $P(0, 0)$ has no dependencies
- Start at $(0, 0)$
- Fill row by row



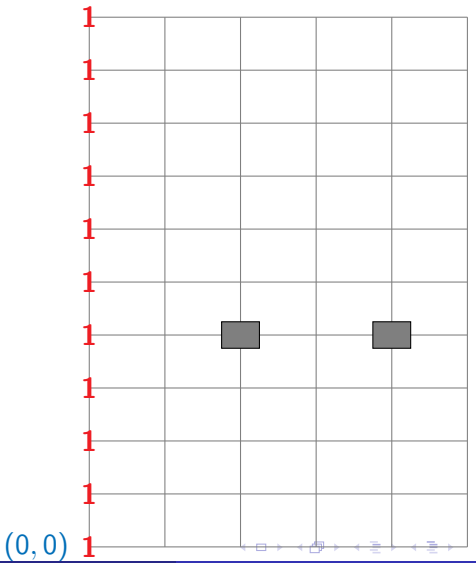
Dynamic programming

- Identify suproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column



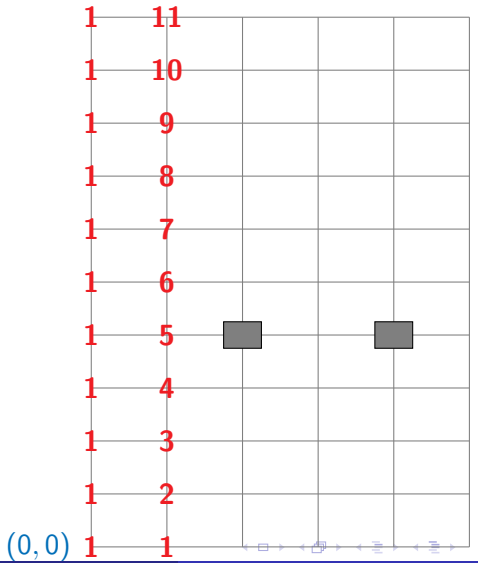
Dynamic programming

- Identify suproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column



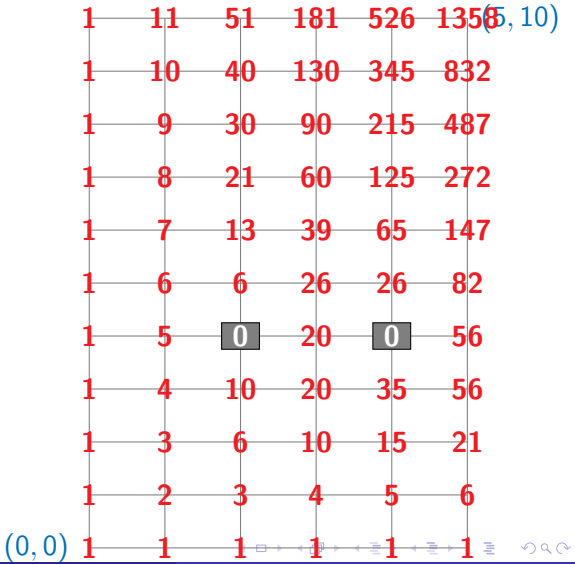
Dynamic programming

- Identify suproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column



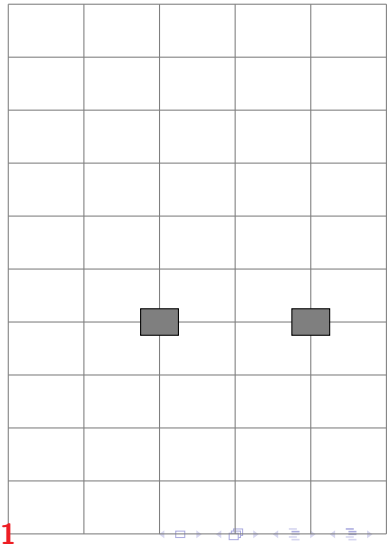
Dynamic programming

- Identify suproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column



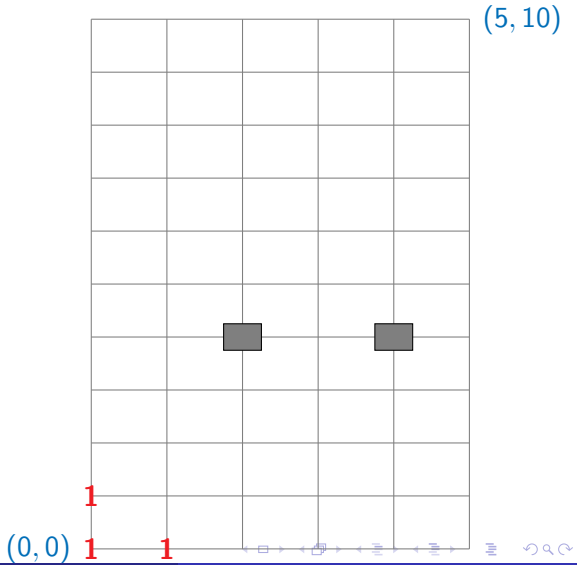
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



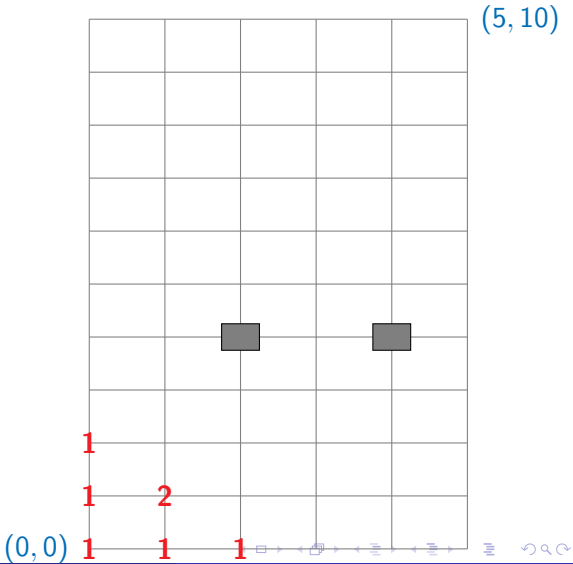
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



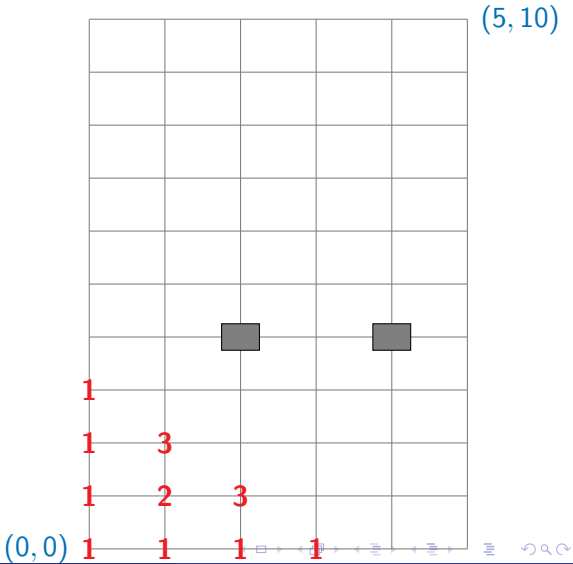
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



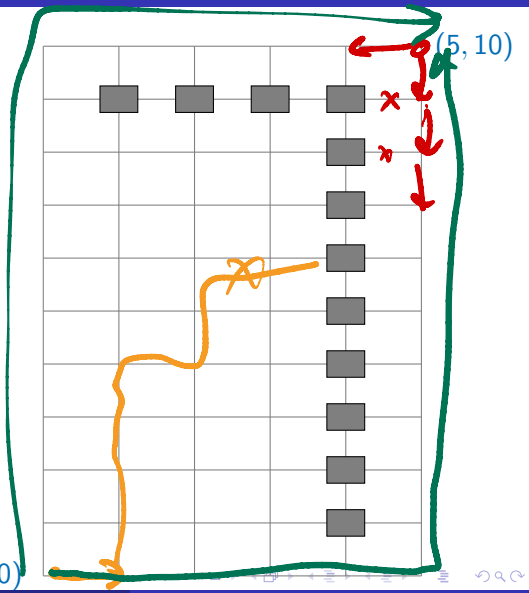
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



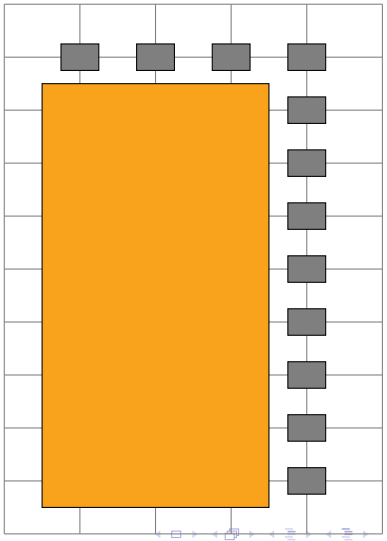
Memoization vs dynamic programming

- Barrier of holes just inside the border



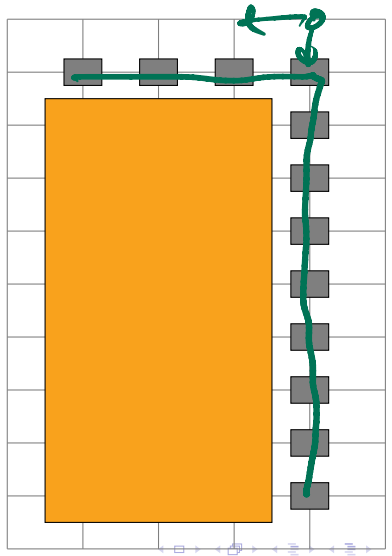
Memoization vs dynamic programming

- Barrier of holes just inside the border
- Memoization never explores the shaded region



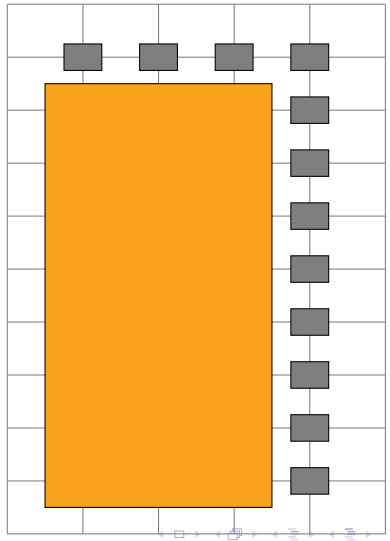
Memoization vs dynamic programming

- Barrier of holes just inside the border
- Memoization never explores the shaded region
- Memo table has $O(m+n)$ entries



Memoization vs dynamic programming

- Barrier of holes just inside the border
- Memoization never explores the shaded region
- Memo table has $O(m+n)$ entries
- Dynamic programming blindly fills all mn cells of the table



Memoization vs dynamic programming

- Barrier of holes just inside the border
- Memoization never explores the shaded region
- Memo table has $O(m+n)$ entries
- Dynamic programming blindly fills all mn cells of the table
- Tradeoff between recursion and iteration
  - "Wasteful" dynamic programming still better in general